Machine Learning Engineer Nanodegree

Capstone Project

Aggelos Papoutsis
May 21, 2019

I. Definition

Project Overview

The project is proposed from the fashion industry. The fashion industry is valued at 385.7 billion dollars[1]. Computer vision and deep learning techniques play a big role in this industry. Nowadays one of the bigger trends is models that can watch the unique preferences of one customer and launch specific recommendation on him. By this way, one company is able to engage relationships with one customer and augmented the possibilities of sales. In this direction an example and one of the bigger trends is Amazon echo which is a artificial intelligence machine that can give to anyone personal fashion advices of what to wear in terms of colors, sizes etc[2].

Fashion-MNIST is a dataset of Zalando's[3] article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

Fashion-MNIST has been addressed in academic portals such as Google Scholar [4] quite in much detail after its origin in 2017. Today as I write this report there exist 759 results in Google Scholar. The paper from Xiao et al[5] is by far the most cited from a variety of researchers[6]. The paper introduced the fashion-MNIST dataset to the world and launch different experiment in order to show the test accuracy as compared to the MNIST benchmark. Beyond this, in the github page of the dataset

[1] https://fashionunited.com/global-fashion-industry-statistics/
[2] https://www.amazon.com/Amazon-Echo-Look-Camera-Style-Assistant/dp/B0186JAEWK
[3] https://github.com/zalandoresearch/fashion-mnist
[4] https://scholar.google.gr/scholar?hl=el&as_sdt=2005&sciodt=0%2C5&cites=18179306951552756 4&scipsc=&q=fashion+mnist&oq=
[5] https://arxiv.org/pdf/1708.07747.pdf
[6] https://scholar.google.gr/scholar?cites=17093201473631702655&as_sdt=2005&sciodt=0,5&hl=en

many researchers have presented their results while they use different techniques[7]. The official review page[8] of the dataset which was written on SEP 2018 shows interesting results. Many institutes like Google or IBM research have launch publications. In USA and China belongs the bigger number of them where generative adversarial networks (GANs) have been on the forefront of the research.

To our opinion the presented results are quite impressive and we expect to reach or bypass them with our model.

Problem Statement

In this project I will built a model that can act as a virtual stylish assistant. The virtual stylish assistant can help a retailer to forecast fashion trends and launch targeting marketing campaigns. The problem has different pictures of shoes, dresses, etc and the deep learning algorithm that we will use is going to classify the images into 10 categories (classes). We expect that our model is going to classify almost all images correctly.

So here the model is going to take each of the images and assign them into the corresponding class. We are going to use convolutional neural networks. As we deal with images we must preserve the spatial dependences between pixels. If for example we have pixels of a bag that pixels are independent on the other pixels around it. We must perform some process in order the images get into the neural network as inputs (this is the point where convolution comes in play). We will use feature detectors and then techniques called maxpooling and dropout. At the end we will flatten and fit them into the neural network.

We will approach the solution with the follow steps

1. Problem statement,
2. Importing data and libraries,
3. Visualize the dataset,
4. Training the model,
5. Evaluating the model.

For the problem, it is useful to use neural networks and particular convolution neural networks. As I say before we have our training data and we will fit them into a neural network but not directly. Here convolutional neural networks begin.

1. So after image preprocessing we have a 28x28 grey image and we will run a convolutional layer,
2. We gonna use feature detectors on our images and built feature maps,

[7] https://github.com/zalandoresearch/fashion-mnist
[8] https://hanxiao.github.io/2018/09/28/Fashion-MNIST-Year-In-Review/

3. Use max polling in order to create smaller sets (reduced feature maps dimensionality),
4. Dropout in order to avoid over fitting,
5. Flatten and finally fit them into a fully connected layer,
6. Use a softmax classifier,
7. After the training we will see the train and test accuracy and get the prediction for the test data,
8. With plotting we will see all classes with the true and predicted values depicted,
9. We will see the true prediction using confusion matrix and classification report.

Metrics

Due to the nature of our dataset (multi class classification with a balanced dataset) for evaluation metrics we can use the precision, recall and F1 score metrics from the classification report. We can see all classes and with our metrics we can see how many of them have been classified correctly.

Precision is a measure of classifier exactness. High precision means that we have low numbers of false positives. In our dataset this is important because we want our classifier to label correctly each sample to the corresponding class and eventually see the proportion of the samples that we have classified correctly. We don't want an image for example of shoes to be classified in the t-shirt class. Instead we want the prediction to match each time the actual class.

High recall means that we have low number of false negatives and can be thought as a measure of classifier completeness. So in our dataset we want our classifier to have low amount of missing results and find all the correct results (in terms of samples). We do not want for example an image of t-shirt to be predicted as not a t-shirt but actually to be a t-shirt. We want the true value (let's called t-shirt to be the true value) to be predicted as often as possible as a t-shirt.

F1 score is a trade (the harmonic mean) between them where 1 is the best result and 0 the worst result.

The official documentation of sciKit-learn[9] describe the metrics that we will use.

<< Compute precision, recall, F-measure and support for each class

[9] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html

The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of beta. beta == 1.0 means recall and precision are equally important.
The support is the number of occurrences of each class in y_true.>>

Also from the official documentation of scikit-learn for the classification metrics[10] we can see the follow.

<<In a binary classification task, the terms ''positive'' and ''negative'' refer to the classifier's prediction, and the terms ''true'' and ''false'' refer to whether that prediction corresponds to the external judgment (sometimes known as the ''observation''). Given these definitions, we can formulate the following table:>>

	Actual class (observation)	
Predicted class (expectation)	*tp (true positive) Correct result*	*fp (false positive) Unexpected result*
	fn (false negative) Missing result	*tn (true negative) Correct absence of result*

In this context, we can define the notions of precision, recall and F-measure:

$$precision = tp/tp+fp,$$

$$recall = tp/tp+fn,$$

$$F\beta=(1+\beta2)precision\times recal/l\beta2precision+recall>>$$

[10] https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics

II. Analysis

Data Exploration

Each image[11] is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels, and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.

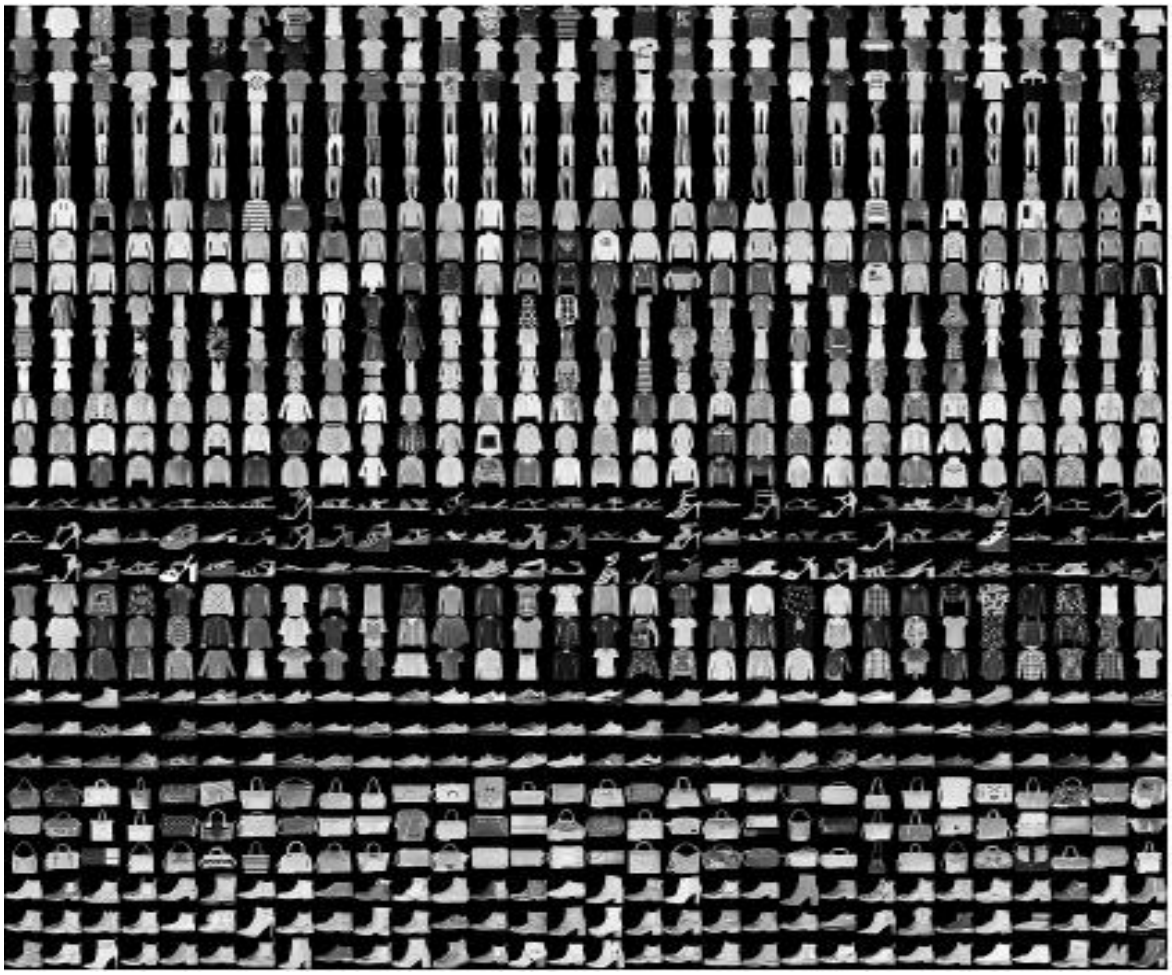

Fig. 1: How the data looks (derived from https://github.com/zalandoresearch/fashion-mnist)

[11] https://github.com/zalandoresearch/fashion-mnist

- To locate a pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between 0 and 27. The pixel is located on row i and column j of a 28 x 28 matrix.
- For example, pixel 31 indicates the pixel that is in the fourth column from the left, and the second row from the top, as in the ascii-diagram below.

Labels

Each training and test example is assigned to one of the following labels:

- T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot

TL;DR

- Each row is a separate image, Column 1 is the class label, Remaining columns are pixel numbers (784 total), Each value is the darkness of the pixel (1 to 255)

Bellow we can see the dimension of our data that we calculate in the Jupyter notebook.

```
#The dimmension of the original train, test set are as following:

print("Fashion MNIST train -  rows:",fashion_train_df.shape[0]," columns:", fashion_train_df.shape[1])
print("Fashion MNIST test  -  rows:",fashion_test_df.shape[0]," columns:", fashion_test_df.shape[1])
```

```
Fashion MNIST train -  rows: 60000  columns: 785
Fashion MNIST test  -  rows: 10000  columns: 785
```

Fig. 2: Dimension of the dataset

Exploratory Visualization

The figure below shows the distribution of the target class in the training. The dataset seems to be high balanced. So there is no need for additional techniques and we can dive directly to the implementation.

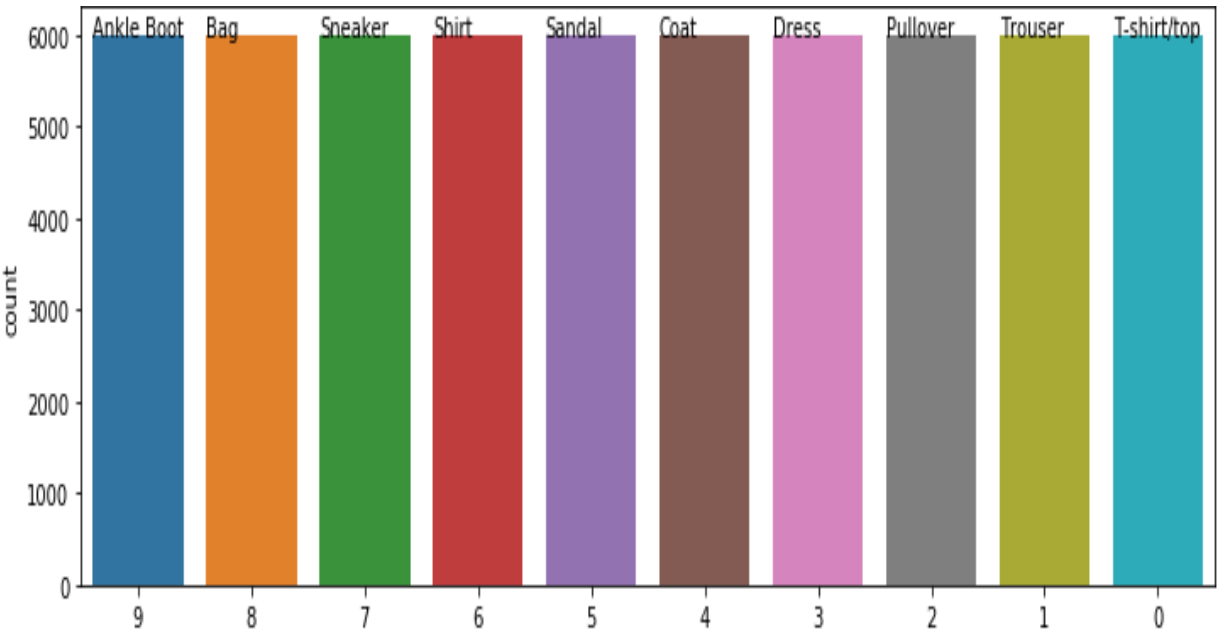

Fig. 3: Number of label for each class

Algorithms and Techniques

As we disgusted before for this problem we will use convolutional neural networks[12]. These types of networks have been recognized as the best solution when we have to deal with lots of images[13]. The images from our dataset cannot feed directly to the network because we have to preserve the spatial dependences between pixels (see Data Preprocessing). For better intuition we will talk about the operations that take place behind the CNNs. Our main purpose is to describe as good as possible the whole procedure and so no to behave to CNNs like black boxes.

To begin with CNNs are like neural networks (a fully explanation of this neural networks is beyond the scope of this report). They consist from neurons and learnable weights and bias. Each neuron takes several inputs, calculates a weighted sum over them, pass it throw an activation function and returns an output. This network has also a loss function. So why CNNs are different from NNs? One main different which is important for our dataset is that the NN takes as input a vector but CNNs can take as input a multi channeled image. For sure neural networks will work for our problem and understand the mapping between our train and test dataset but it cannot understand the patterns. For example let's say that we have an image of

[12] https://keras.io/layers/convolutional/
[13] http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

shoes that are in half under the bed. CNNs will "see" the shoes as it goes deep but not the NNs. So we don't have to give features. CNNs will understand the right features by themselves.

So a CNN will find the spatial structure of the images and learn in order to produce something useful as output. For an image of a t-shirt for example the network will learn some local features like shape etc. This procedure is learnt in the **convolution layers** which are the first building block of CNNs. These layers consist of many **kernels** (sometimes called **convolution filters**) and their job is to learn the local features that present in an image. A local feature that the convolution layer learns is called **feature map**. Then these features are convolved over the image and produce a matrix that called **activation map.** This map gives a high value in a location if the feature represented in the convolution filter is present in the location of the input.

The second building block is called **pooling layers** which make the features that have been learnt from the CNN invariant. There are two different mechanisms here called max pooling and average pooling. In our project we use max pooling.

The third building block is called fully connected layers. These layers will aggregate the learnt features from different kernels and build a representation of the image. The neurons in a fully connected layer will activated only when different aspects that represented in the convolution features are presented in the input. So the activated neurons will generate different patterns based on features that are presented in the input image make a fully representation of what exactly exist and the image and pass it to the output layer in order to accomplice a correct classification of the image.

So to summarize this convolution layers will learn various local features in the data (e.g. how a T-shirt looks like), pooling layers will make the CNN invariant to translations of these features (e.g. If the local features of a t-shirt slightly translated the CNN will still recognize) and the fully connected layer will say that I find three different local features and with my knowledge it must be at-shirt and will activate the correct output.

The different features that consist a t-shirt do not appear randomly. The main purpose is to be learnt given data. We must define a cost function that gives reward to correctly identified data and penalized the misclassified data. For our project the loss function is sparse categorical cross entropy. In fact we optimize each convolution layer and fully connected neurons by implementing gradient back propagation.

Benchmark

For the benchmark model, we used a simple two layer neural network (fig. 3). I compiled this with the same loss function and optimizer as the main model. The model works well but not as much as we can achieve with the use of convolutional networks. We achieved accuracy bellow 90% and many classes classified incorrectly.

```
model = Sequential([
    Dense(512, input_shape=(784,), activation='relu'),
    #Dense(128, activation = 'relu'),
    Dense(10, activation='softmax')])
```

Fig. 4: Two layer neural network

III. Methodology

Data Preprocessing

Based on the Data Exploration section, there were no abnormalities or characteristics that needed to be addressed. The distribution of the target class in the training seems to be highly balanced. All classes seem to have same amount of data so in the beginning of our problem there is no need for any preprocessing.

Before we dive into the implementation we perform some transformations. First of all keras[14] wants the data frames to be transformed into arrays. After we do that we take into account that many machine learning models works better, if the input data been normalized [0, 1] or [-1, 1]. So we want our pixel data to be between 0-1 and not between 0-255 (the number measures the intensity of the corresponding pixel). So we rescale our x_train and x_test pixel data by divided each pixel value with 255.

As we have already discussed we cannot pass directly the images into our network. So we must reshape the sizes of the columns (784) to the original size of the images (28χ28χ1). The 784 comes because the data have been flattened each images into a single dimension of 784 as many machine leaning models expect each training example to be a vector.

[14] https://keras.io/

Implementation

Bellow we can see the structure of our model. We used a sequential model which is a linear stack of layers. At first it needs to initialized and then we can add layers with the add method.

```
cnn_model = Sequential()


cnn_model.add(Conv2D(32,3, 3, input_shape = (28,28,1), activation='relu'))
cnn_model.add(MaxPooling2D(pool_size = (2, 2)))
cnn_model.add(Dropout(0.2))
cnn_model.add(Conv2D(64,3, 3, input_shape = (28,28,1), activation='relu'))
cnn_model.add(MaxPooling2D(pool_size = (2, 2)))
cnn_model.add(Dropout(0.2))
cnn_model.add(Conv2D(128,3, 3, input_shape = (28,28,1), activation='relu'))
cnn_model.add(Dropout(0.2))
cnn_model.add(Flatten())
cnn_model.add(Dense(output_dim = 128, activation = 'relu'))
cnn_model.add(Dropout(0.2))
cnn_model.add(Dense(output_dim = 10, activation = 'softmax'))
```

Fig. 5: Structure of the model

After this we used a convolutional layer and we specified the filters (32) and the kernel size (3x3), along with the input shape which is the size of the image that presented to the CNN and the activation function which is the rectified linear unit (relu). We then used a Maxpooling2D which is a Max pooling operation for spatial data and we specified the pool size (2, 2) which shows as the factors that we will downscale in both directions.

After that a dropout layer has been used in order to avoid over fitting. Then we use a second convolutional layer with 64 filters and also use the maxpolling and dropout techniques as above. Then we use a third convolutional layer with 128 filters and also use the maxpolling and dropout techniques as above. After that we flattened the layer in order to fit in a fully connected layer which has 128 nodes and relu as activation function. The second dense layer has 10 output and a softmax activation function which is consider to be better than sigmoid in our problem. Bellow we can see the summary of our model.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 128)         73856
_________________________________________________________________
dropout_3 (Dropout)          (None, 3, 3, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               147584
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
```

Fig. 6: Summary of the model

Before the training part is necessary to configure the learning process. We will do that by compiling our CNN model. We want three arguments as presented in the original keras documentation about sequential models[15].

- *<< An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the `Optimizer` class..*
- *A loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as `categorical_crossentropy` or `mse`), or it can be an objective function..*
- *A list of metrics. For any classification problem you will want to set this to `metrics=['accuracy']`. A metric could be the string identifier of an existing metric or a custom metric function.>>*

[15] https://keras.io/getting-started/sequential-model-guide/

The loss function is used to measure how accuracy is the model during training. Our goal is to minimize the loss function in order to make the model more robust. Optimizer is used in order to see how the model is updated based on the data and the loss function. Metrics are used to monitor the training and testing steps. For our model we use sparse categorical crossentropy as loss function[16], Adam[17] as optimizer and accuracy as metric. The previous loss function was used because our targets are integer. Adam was used as seems that work perfect for large datasets[18]. The default learning rate is 0.001. Accuracy is used by default in any classification problem and is used in order to see the images that are classified correctly.

After that we train our model with the fit function[19] using our x,y numpy arrays, and we specified the number for the gradient update (batch_size) and the number of times that we iterate over x,y (epochs). Finally we use our validation data.

The procedure was very straightforward for us and during the experiments we did not encounter any difficult situation in terms of coding or the overall procedure. I think this happens because Keras is a well documented library and images based datasets have been well analyzed over the past years.

Refinement

For this project I used three different architectures in order to see the affect on my model. In the beginning I used 1 convolutional network. Then I used a model with 2 convolutional networks and final I used three convolutional networks. I implement the models in three different Jupyter notebooks for more clarity. The model with the three CNNs seems to works better as we will present in the result section than the previous two.

IV. Results

Model Evaluation and Validation

After the training part we will discussed the results for the three different models that we used in order to see how to model works with different architectures. In the

[16] https://keras.io/losses/
[17] https://keras.io/optimizers/
[18] https://arxiv.org/abs/1412.6980v8
[19] https://keras.io/models/sequential/

beginning we used a one layer CNN model. The train accuracy after 50 epochs was 0.975.

```
evaluation = cnn_model.evaluate(X_train, y_train)
print('Train Accuracy : {:.3f}'.format(evaluation[1]))
```

```
48000/48000 [==============================] - 20s 418us/step
Train Accuracy : 0.975
```

Fig. 7: Train accuracy of one layer CNN

The test accuracy was 0.918. The correct predicted classes were 9176 and the incorrect predicted classes were 824.

```
#how the model performs on the test dataset:

evaluation = cnn_model.evaluate(X_test, y_test)
print('Test Accuracy : {:.3f}'.format(evaluation[1]))
```

```
10000/10000 [==============================] - 4s 440us/step
Test Accuracy : 0.918
```

Fig. 8: Test accuracy of one layer CNN

Then we used a two layer CNN model. The train accuracy after 50 epochs was 0.947.

```
evaluation = cnn_model.evaluate(X_train, y_train)
print('Train Accuracy : {:.3f}'.format(evaluation[1]))
```

```
48000/48000 [==============================] - 24s 492us/step
Train Accuracy : 0.947
```

Fig. 9: Train accuracy of two layer CNN

The test accuracy was 0.923. The correct predicted classes were 9230 and the incorrect predicted classes were 770.

```
#how the model performs on the test dataset:

evaluation = cnn_model.evaluate(X_test, y_test)
print('Test Accuracy : {:.3f}'.format(evaluation[1]))
```

```
10000/10000 [==============================] - 4s 439us/step
Test Accuracy : 0.923
```

Fig. 10: Test accuracy of one layer CNN

Our final model was created with three CNN layers. Here the train accuracy was 0.972.

```
evaluation = cnn_model.evaluate(X_train, y_train)
print('Train Accuracy : {:.3f}'.format(evaluation[1]))
```

```
48000/48000 [==============================] - 27s 568us/step
Train Accuracy : 0.972
```

Fig. 11: Train accuracy of three layer CNN

The test accuracy was 0.929. The correct predicted classes were 9287 and the incorrect predicted classes were 713.

```
#how the model performs on the test dataset:

evaluation = cnn_model.evaluate(X_test, y_test)
print('Test Accuracy : {:.3f}'.format(evaluation[1]))
```

```
10000/10000 [==============================] - 9s 908us/step
Test Accuracy : 0.929
```

Fig. 12: Test accuracy of three layer CNN

As we can see from the follow table our final model give better results.

	Train Accuracy	Test Accuracy	Correct Predicted Classes	Incorrect Predicted Classes
1 CNN model	0.975	0.918	9.176	824
2 CNN model	0.947	0.923	9.230	770
3(final) CNN model	**0.972**	**0.929**	**9.287**	**713**

Table 1: Results of the three CNN models

Our final model seems to work fine as we did not realize any signs of over fitting. The validation accuracy increases after few epochs and the validation loss is decreasing.

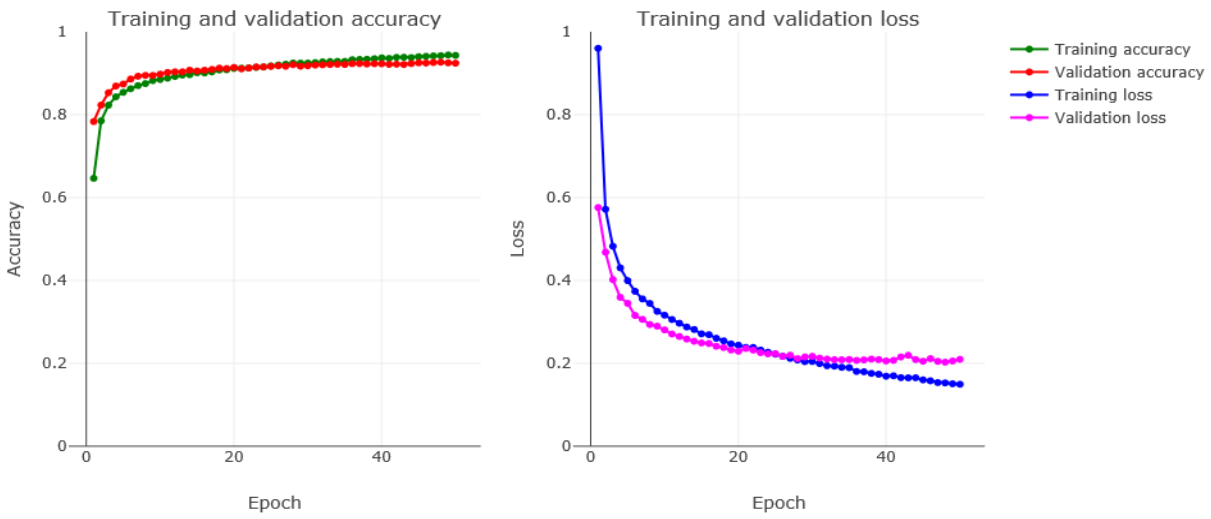

Fig. 13: Validation accuracy and loss

So after all this analysis we can present the final results from confusion matrix and classification report. Before this and in order to be more clear we will present the classes and the content of each one of them.

0: **T-shirt/top**, **1**: **Trouser**, **2**: **Pullover**, **3**: **Dress**, **4**: **Coat**, **5**: **Sandal**, **6**: **Shirt**, **7**: **Sneaker**, **8**: **Bag**, **9**: **Ankle boot**.

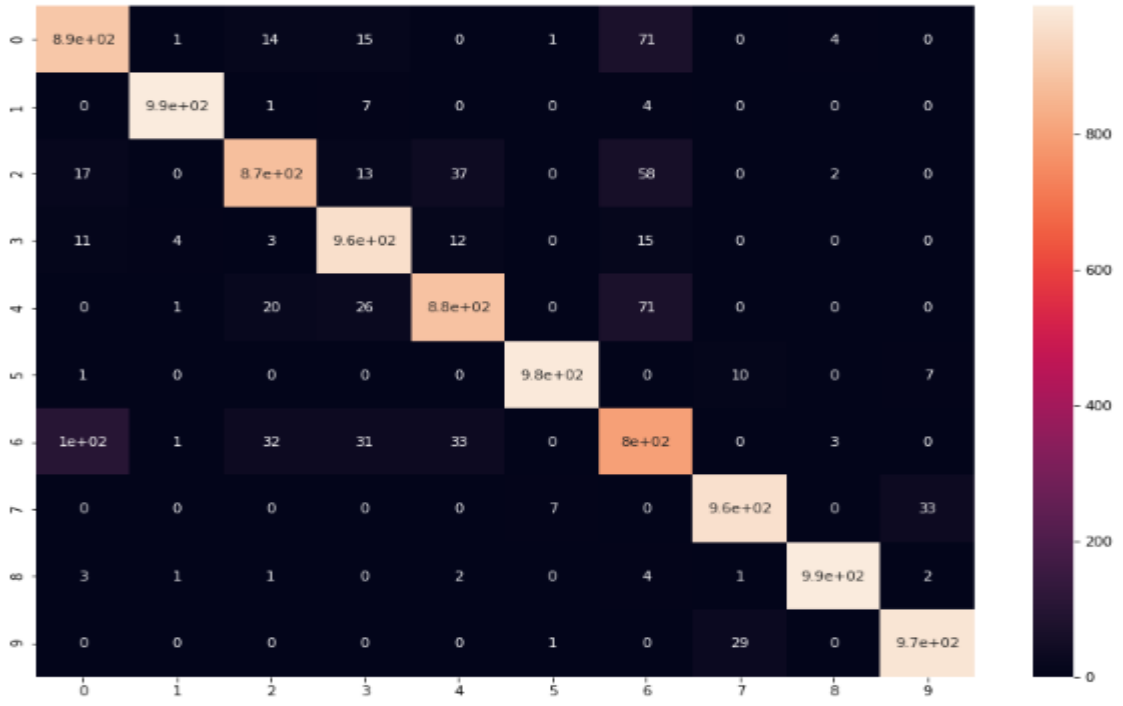

Fig. 14: Confusion matrix

	precision	recall	f1-score	support
Class 0	0.87	0.89	0.88	1000
Class 1	0.99	0.99	0.99	1000
Class 2	0.92	0.87	0.90	1000
Class 3	0.91	0.95	0.93	1000
Class 4	0.91	0.88	0.90	1000
Class 5	0.99	0.98	0.99	1000
Class 6	0.78	0.80	0.79	1000
Class 7	0.96	0.96	0.96	1000
Class 8	0.99	0.99	0.99	1000
Class 9	0.96	0.97	0.96	1000
micro avg	0.93	0.93	0.93	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

Fig. 15: Classification report

As we can see for class 1 (trouser), class 5 (sandal), and class 8 (bag) our network was 99% accurate! For the other classes the network was over 90% accurate. The only problem was with class 6 (shirt). There the network seems to struggle and as can see from the confusion matrix many samples have been classified incorrectly. Probably this happen because the images of shirt are not very clear and the network easily can "see" them for example like pullovers.

In our final model we manual specify the dataset for validation during training. We use the train_test split and use 20 % of the data for validation and 80% data for training. Inside the fit method we specify the validation dataset with the validation_data argument. In order to investigate the performance of our model with different methods we create the same model, but this time we set the validation_split argument inside the fit function. The performance was pretty much the same.

We also try to add some Gaussian noise after the relu function like a noisy activation function and after the polling layer in order to see the performance of our model. We do not realize any sights of improving or any signs of overfitting. In general this model works as good as our final one.

More of this the problem with deep learning model is that are stochastic. Artificial NN use randomness as they fit on the dataset. So we have random initial weights and random shuffling of data each time an epoch run. So each time the model fit on the same data we may end up with a different prediction. We run 5 times our model and also we try to run the model without to define the random state. In all cases the results was pretty much the same as our final model. So after all this tests I think that the final model is robust enough for our experiment.

	Train accuracy	Test accuracy	Correct Predicted Classes	Incorrect Predicted Classes
Final model	**0.972**	**0.929**	**9.287**	**713**
1st Run	0.970	0.926	9.261	739
2 run	0.969	0.929	9.287	713
3rd run	0.971	0.925	9.253	747
4 run	0.973	0.929	9294	706
5 run	0.971	0.930	9.297	703
Without random state	0.971	0.927	9.273	727
With noise	0.952	0.917	9.168	832
With automatic verification dataset	0.954	0.927	9.271	729

Table 2: Results of models

Justification

Here we will compare the final model with the benchmark model. As we can see the benchmark model with the 2 neural network layers performs quite badly. After the CNN layers and the additional techniques that we use we achieve a much better model.

	Train Accuracy	Test Accuracy	Correct Predicted Classes	Incorrect Predicted Classes
3(final) CNN model	**0.972**	**0.929**	**9.287**	**713**
Benchmark model	0.979	0.899	8.987	1.013

Table 3: Final model VS Benchmark model

Also as can see from the follow figure the benchmark model tends to overfit.

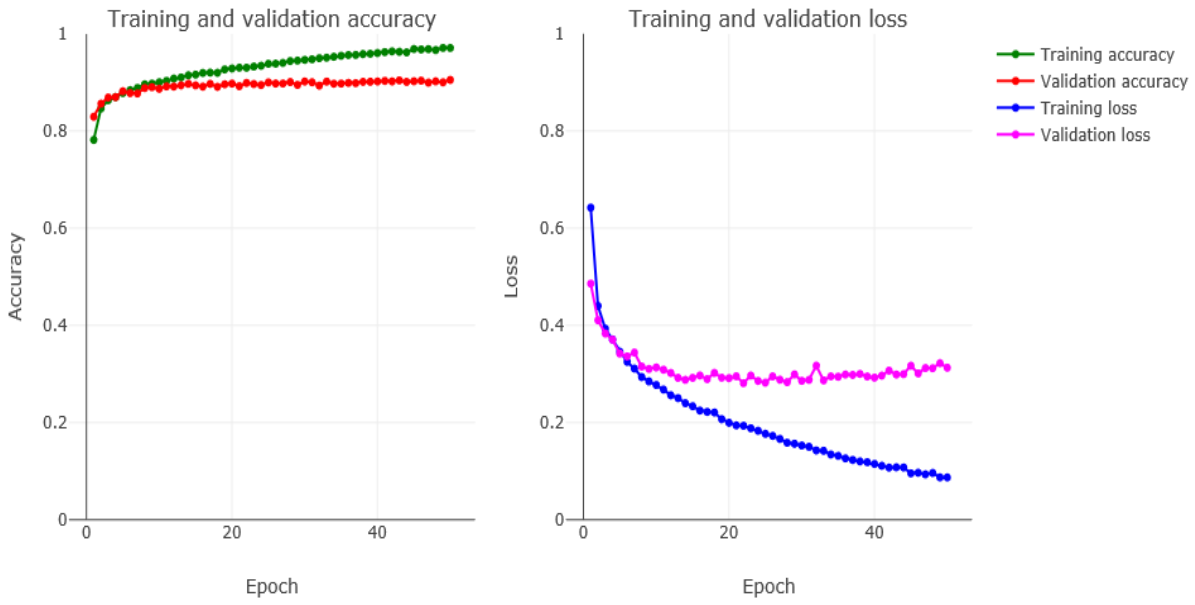

Fig. 16: Validation accuracy and loss

We can see also from classification report that the benchmark model have not done as good work as our final model with many classes be accurate down 90%.

	precision	recall	f1-score	support
Class 0	0.81	0.89	0.85	1000
Class 1	0.99	0.98	0.98	1000
Class 2	0.83	0.84	0.83	1000
Class 3	0.90	0.91	0.91	1000
Class 4	0.85	0.85	0.85	1000
Class 5	0.99	0.94	0.96	1000
Class 6	0.79	0.70	0.74	1000
Class 7	0.93	0.94	0.94	1000
Class 8	0.97	0.97	0.97	1000
Class 9	0.93	0.97	0.95	1000
micro avg	0.90	0.90	0.90	10000
macro avg	0.90	0.90	0.90	10000
weighted avg	0.90	0.90	0.90	10000

Fig. 17: Classification report of the benchmark model

V. Conclusion

Free-Form Visualization

Bellow we can see the true and predicted classes from our dataset. As we talk before the network seems to be confused in same images but in general I think that it does a very nice job.

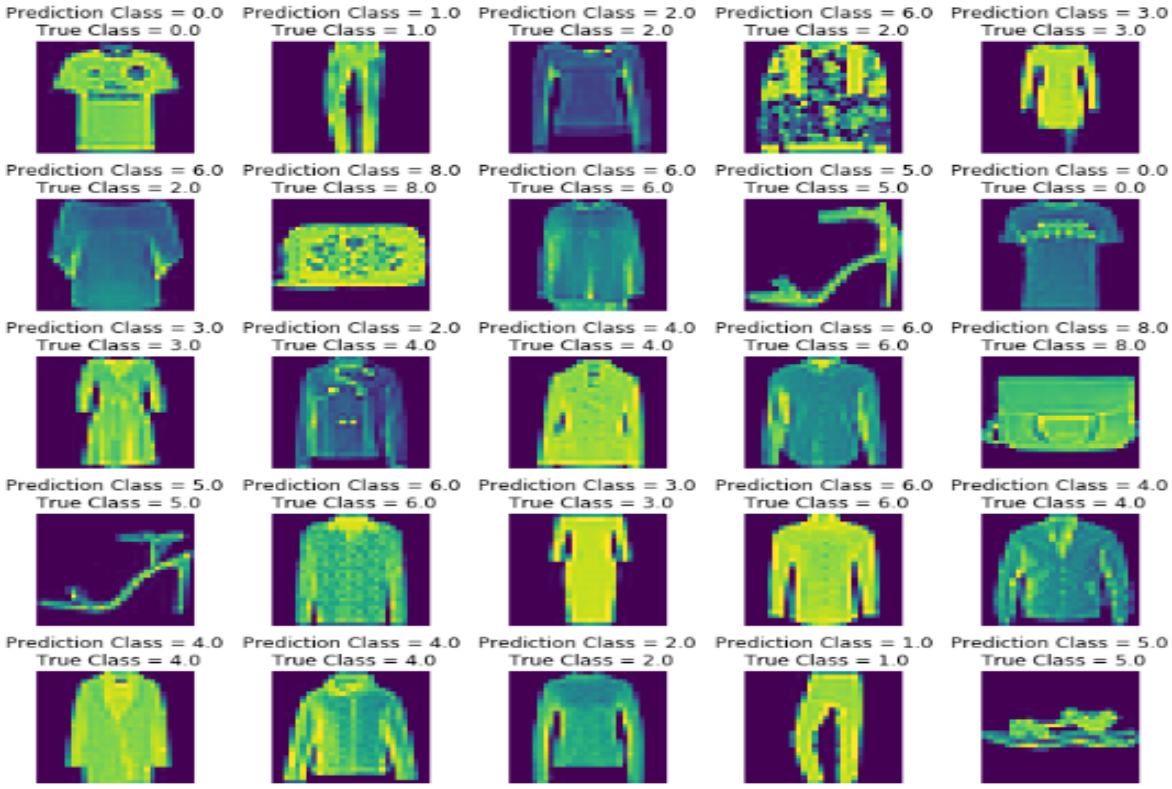

Fig. 18: True and predicted classes

Reflection

In this section we can talk again the steps that we use in order to specify and solve our problem.

We approach the solution with the follow steps:

- Problem statement,
- Importing data and libraries,
- Visualize the dataset,
- Training the model,
- Evaluating the model.

For the problem, we use deep neural networks and particular convolution neural networks. The procedure was the follow.

- After image preprocessing we have a 28x28 grey image and we ran a convolutional layer,
- We used feature detectors on our images and built feature maps,

- We used max polling in order to create smaller sets (reduced feature maps dimensionality),
- Dropout in order to avoid over fitting,
- Flatten and finally fitted them into a fully connected layer,
- Used a softmax classifier,
- After the training we calculate the train and test accuracy and got the prediction for the test data,
- With plotting we depicted the classes with the true and predicted values,
- We investigated the accurate of our model using confusion matrix and classification report.

I have not had difficult time with the project. The keras documentation is very straightforward and to my point of view Keras is a fantastic tool to work with. What is more fashion industry is a high value sector that will benefit a lot in the future from artificial intelligence and machine learning.

Finally as derive from our results the network did a nice job and fit our expectation. But I think that it cannot be used directly for real world fashion problems. The fashion MNIST dataset is a replacement of MNIST dataset which means that the data have already been processed. In order to use fashion images we must perform preproccesing steps as the Fashion MNIST dataset and maybe use techniques such as multi-label classification and multi-output networks.

Improvement

Our model achieves accuracy of 0.929 which as compared to the models on the github page[20] of the dataset is very promising. For sure if we use data augmentation techniques or different combination of CNN networks and fully connected networks we will have better results. What is more I did not have the change to perform grid search in order to find the best parameters due to the luck of resources but I think that I used enough good metrics and hyperparameters.

[20] https://github.com/zalandoresearch/fashion-mnist

Classifier	Preprocessing	Fashion test accuracy	MNIST test accuracy	Submitter	Code
2 Conv+pooling	None	0.876	-	Kashif Rasul	
2 Conv+pooling	None	0.916	-	Tensorflow's doc	
2 Conv+pooling+ELU activation (PyTorch)	None	0.903	-	@AbhirajHinge	
2 Conv	Normalization, random horizontal flip, random vertical flip, random translation, random rotation.	0.919	0.971	Kyriakos Efthymiadis	
2 Conv <100K parameters	None	0.925	0.992	@hardmaru	
2 Conv ~113K parameters	Normalization	0.922	0.993	Abel G.	
2 Conv+3 FC ~1.8M parameters	Normalization	0.932	0.994	@Xfan1025	
2 Conv+3 FC ~500K parameters	Augmentation, batch normalization	0.934	0.994	@cmasch	
2 Conv+pooling+BN	None	0.934	-	@khanguyen1207	
2 Conv+2 FC	Random Horizontal Flips	0.939	-	@ashmeet13	
3 Conv+2 FC	None	0.907	-	@Cenk Bircanoğlu	
3 Conv+pooling+BN	None	0.903	0.994	@meghanabhange	
3 Conv+pooling+2 FC+dropout	None	0.926	-	@Umberto Griffo	
3 Conv+BN+pooling	None	0.921	0.992	@GunjanChhablani	
5 Conv+BN+pooling	None	0.931	-	@Noumanmufc1	
CNN with optional shortcuts, dense-like connectivity	standardization+augmentation+random erasing	0.947	-	@kennivich	
GRU+SVM	None	0.888	0.965	@AFAgarap	
GRU+SVM with dropout	None	0.897	0.988	@AFAgarap	
WRN40-4 8.9M	standard preprocessing (mean/std				

Fig. 19: Some results of different architectures from Fashion MNIST